Implementation of a Hardware Coordinate Wise Descend Algorithm with Maximum Likelihood Estimator for Use in mMTC Activity Detection

Mikael Henriksson
Master of Science Thesis in Electrical Engineering

Implementation of a Hardware Coordinate Wise Descend Algorithm with Maximum Likelihood Estimator for Use in mMTC Activity Detection:

Mikael Henriksson
LiTH-ISY-EX–20/5326–SE

Supervisor:  Oscar Gustafsson  
ISY, Linköpings universitet

Examiner:  Mattias Krysander  
ISY, Linköpings universitet

Division of Computer Engineering
Department of Electrical Engineering
Linköping University
SE-581 83 Linköping, Sweden

Copyright © 2020 Mikael Henriksson
Abstract

In this work, a coordinate wise descent algorithm is implemented which serves the purpose of estimating active users in a base station/client wireless communication setup. The implemented algorithm utilizes the sporadic nature of users, which is believed to be the norm with 5G Massive MIMO and Internet of Things, meaning that only a subset of all users are active simultaneously at any given time. This work attempts to estimate the viability of a direct algorithm implementation to test if the performance requirements can be satisfied or if a more sophisticated implementation, such as a parallelized version, needs to be created.

The result is an isomorphic ASIC implementation made in a 28 nm FD-SOI process, with proper internal word lengths extracted through simulation. Some techniques to lessen the burden on hardware without losing performance is presented which helps reduce area and increase speed of the implementation. Finally, a parallelized version of the algorithm is proposed, if one should desire to explore an implementation with higher system throughput, at almost no further expense of user estimation error.
Acknowledgments

A huge thank you to my supervisor Oscar Gustafsson, who on multiple occasions helped me back on the right track when I was lost and who always delivered fantastic feedback.

I would also like to thank Petter Källström and Frans Skarman for letting me bug you with my stupid ideas.

Linköping, May 2020
# Contents

1 Introduction
   1.1 Motivation ........................................... 2
   1.2 Purpose ............................................ 2
   1.3 Research issue ..................................... 2
   1.4 Delimitations ....................................... 3

2 Theory
   2.1 Notation ............................................. 5
   2.2 Terminology ........................................ 5
      2.2.1 Performance terms .............................. 6
      2.2.2 Hardware terms ................................. 6
      2.2.3 Other terms .................................... 6
   2.3 Studied scenario ................................... 7
   2.4 The problem ....................................... 8
   2.5 Algorithm to implement ............................ 9
      2.5.1 Updating the variance estimate ............... 9
      2.5.2 Performance requirements ........................ 9
   2.6 Hardware aspects of implementation ............... 10
      2.6.1 Numeric representation ........................ 10
      2.6.2 Word length .................................... 10
      2.6.3 Combinational latency .......................... 11
      2.6.4 Critical path .................................. 11
      2.6.5 Critical loop .................................. 11
      2.6.6 Isomorphic implementation ..................... 13

3 Results of isomorphic implementation ................. 15
   3.1 Models ............................................ 15
      3.1.1 MATLAB model ................................ 16
      3.1.2 C++ model using floating-point arithmetic ....... 17
      3.1.3 C++ model using fixed-point arithmetic ........ 18
      3.1.4 RTL VHDL model ................................ 18
   3.2 Sequence ........................................... 18
      3.2.1 Random sequences with memories ............... 18
Introduction

A key feature of future wireless communication systems, such as the Internet of Things, is believed to be few and sporadic user patterns where only a small subset of all possible users require communication resources at a given time [5, 8]. In such scenarios, of massive connectivity communication, the transfer of data between a Base Station and a Client has two phases. One phase of detecting the current set of active users and estimating a channel for those users with only a fraction of the available communication resources (illustrated in Figure 1.1), and another phase of transmitting data over the remaining available resources. This two phase operation scheme is the base of current, and likely future, wireless communication systems.

A recently proposed algorithm for activity detection, the procedure of finding the subset of active users among all possible users, in Massive MIMO communication systems is based on a coordinate wise descent with a maximum likelihood estimator to assay the subset of active users [2, 3, 5]. In a scenario where the set of possible users, the number of base station antennas and the length of pilot sequences are large, the amount of computations needed to perform these algorithms is also very large. Even more importantly, since the procedure of activity

---

Figure 1.1: Illustration of a base station and some client users.
detection in a wireless communication system should only use a fraction of the available communication resources, coherence time being one such resource, the demand for low latency execution is of importance.

To the authors knowledge, there are only two earlier publications [11, 13] implementing activity detection algorithms for use in massive machine type communication. However, their use case are somewhat different from this implementation, making it difficult to directly compare performance between these implementations.

1.1 Motivation

The gap between a mathematical representation of an algorithm and its hardware implementation is not always trivial to bridge. The sheer amount of arithmetic computations and the algorithms degree of parallelism can limit the desired performance of an algorithmic implementation. By attempting to implement the mathematical algorithm in hardware, the difficulties of implementation will manifest themselves which have the possibility to extend the bridge between mathematically rigorous algorithms and implementation-wise reasonable algorithms. Figure 1.2 illustrates the process of taking a mathematical representation of an algorithm to tape out of an application specific integrated circuit.

1.2 Purpose

This work serves the purpose of making a draft implementation of a new activity detection algorithm (Chapter 2, Algorithm 1) for uses in massive machine type communication systems (mMTC), which can be of help in the upcoming release of the fifth generation telecommunications systems. The implementation serves its purpose of helping bridge the implementation gap, specified in the Motivation, Section 1.1.

1.3 Research issue

Given the coordinate descent algorithm at hand (Chapter 2, Algorithm 1), how can such an algorithm be implemented on an application specific integrated circuit (ASIC)? With this for a base we specify more specifically what should be answered.

- What is the algorithmic performance of the implemented algorithm, measured in appropriate metric, probability of miss detect and probability of false alarm (defined under Section 2.4)?
- How many client devices does the implementation support?
- What is the power consumption of such an ASIC?
• Is it feasible to implement the algorithm as it is described in Algorithm 1 on an ASIC? If not, what could make the algorithm more suitable for implementation without major degradation of performance?

• If the ASIC can not be successfully implemented, what did go wrong? What could/should have been done different in order to succeed?

![Figure 1.2: Flowchart showing the process of taking an algorithm described mathematically to one of its corresponding hardware implementation.](image)

1.4 Delimitations

The given scenario, which is described under Section 2.3, requires equipment for transmitting and receiving wireless data, AD/DA converters and possibly more equipment in order to operate. The entry point for this solution will be the digital matrix with received signals which is described under Section 2.3. Anything involving the creation of such a matrix, the transfer of signals and conversion between digital and analog domain, will not be discussed in this work.

As the focus of this work is on the hardware implementation aspect of the algorithm, we deliberately leave the work of comparing performance of this activity detection algorithm to other closely related algorithms and mainly focus on the degradation of this algorithm throughout the implementation flow. Curious readers can find more information and comparisons between activity detection algorithms in [3] and [5].
This chapter will describe the theory regarding this project. It will briefly start off by describing the notation and terminology used throughout the work and the studied mMTC scenario as it is proposed in [5] and [2]. It will also present the studied algorithms used for activity detection and introduce the concept of hardware latency and describe how it can impact implementation size (as in area on an implemented circuit), power consumption and algorithmic performance.

2.1 Notation

Scalar values are denoted with italic letters (e.g., $a$ or $B$), vectors are denoted with small boldface letters (e.g., $c$) and matrices are denoted with capital boldface letters (e.g., $D$). The $i$-th row vector of a matrix $D$ is denoted $D_{i,:}$ and the $j$-th column vector of the same matrix is denoted $D_{:,j}$. The Hermitian operation of a complex valued matrix $D$ is denoted $D^H$ and such a matrix satisfy $D_{i,j}^H = \overline{D}_{j,i}, \forall i, j$. The boldface character $I$ is reserved for denoting the identity matrix and its subscript shows the size such that $I_k$ denotes the identity matrix of size $k \times k$.

Further more, sets are denoted with calligraphic letters (e.g., $K$) and the cardinality of a set $K$ is denoted $|K| \in \mathbb{N}$. For a scalar $V \in \mathbb{N}$ we denote the set of natural numbers up to $V$ as $[V] = \{1, 2, 3, ..., V\}$.

2.2 Terminology

Before going much further it would be good to summarize and explain some of the terms and phrases used throughout this document and to clarify what the author means using these terms.
2.2.1 Performance terms

The term algorithmic performance is used to describe algorithm metrics measurable standalone from its implementation, i.e., an algorithm’s statistical probability of producing an erroneous result, its rate of convergence or some asymptotic operation complexity. These metrics can (in many cases) be thought of as separable from the hardware implementation.

The term implementation performance (sometimes system performance or hardware performance) is used to describe performance metrics that is associated with the hardware implementation, i.e., silicon area, power consumption or maximum system clock frequency.

The author realizes that there is a huge overlap between these two very broad terms, but feels it is of importance to specify these umbrella terms to be able to discuss the impact of changes made throughout the implementation process.

2.2.2 Hardware terms

Register-transfer level (RTL) is the top abstraction level (least detailed) digital description of a synchronous digital circuit [14]. It is often the description of a synchronous digital system in some hardware description language, i.e., Verilog or VHDL, and the RTL description serves as the last step in this implementation, from which some synthesis tool will generate the necessary objects for fabricating an application specific integrated circuit (ASIC).

2.2.3 Other terms

Making a binary estimate whether a user is active (true) or inactive (false) is known as binary classification and a system performing such a task is known as a binary classifier. The outcome of binary classification has four states, illustrated in Figure 2.1, of which true positives and true negatives are desired outcomes. A good binary classifier will produce many true positives or negatives, and will produce few false positives or negatives.

Receiver operating characteristic (ROC) is a graphical plot showing a statistical measurement for performance of a binary classifier, with respect to some varying variable. In this project, ROC curves will be used to present the algorithmic performance of the activity detection algorithm by plotting probability

\[
\text{Condition positive} \quad | \quad \text{Condition negative}
\]

| Predicted positive | True positive | False positive |
| Predicted negative | False negative | True negative |

\textbf{Figure 2.1: Illustration of the four states possible in binary classification.}
of false positives against probability of false negatives (both defined under Section 2.4) for a varying threshold parameter (presented later in Section 2.5). For an example of an ROC curve, see the right plot in Figure 3.5.

Whenever a ROC curve is presented throughout the rest of this document, it shows the greatest obtainable performance, that is, the performance when convergence has been reached.

### 2.3 Studied scenario

The scenario studied in this work, is that of a massive MIMO setup with a single base station having $M$ antennas. There exist a set $\mathcal{K}_c$ of $K_c = |\mathcal{K}_c|$ possible users from which the proper subset $\mathcal{A}_c \subseteq \mathcal{K}_c$ are active users, that is, users in need of service from the base station (see Figure 2.2). A block fading channel model [1] is used, i.e., a channel model in which coherence blocks of some adjacent frequency and time are static (has constant attenuation over frequency and is approximately time invariant). Each coherence block consists of $D_c$ signal dimensions. Due to the requirements put on most wireless communication systems, only a few coherence block can be assigned to serve the purpose of activity detection. Further, due to the sporadic nature of users in this model, the set of active users is considered significantly smaller than the set of possible users $\mathcal{A}_c = |\mathcal{A}_c| \ll K_c = |\mathcal{K}_c|$.

The data when transferred over the block fading channel will get slightly distorted and some amount of additive white Gaussian noise will be added to the received signal. The channel for a user $k$ will distort the transferred signal through the $M$-dimensional channel vector $\mathbf{h}_k$ and add noise $\mathbf{z}$ such that the received signal for dimension $i \in [D_c]$ will have the form

$$y[i] = \sum_{k \in \mathcal{K}_c} b_k \sqrt{g_k} a_{k,i} \mathbf{h}_k + z[i] \quad (2.1)$$

where $a_{k,c}$ denotes the pilot sequence of user $k$, where $g_k$ denotes the transmitted signal strength, and where $b_k \in \{0, 1\}$ is the activity of user $k$. The channel vectors $\mathbf{h}_k$ are independent and spatially white i.e., $\mathbf{h}_k \sim \mathcal{CN}(0, \mathbf{I}_M)$.

![Figure 2.2: Illustration of the studied scenario, showing users and the base station.](image-url)
Further, we describe the entire received signal over all signal dimensions as

\[ Y = \mathbf{A} \Gamma^{\frac{1}{2}} \mathbf{H} + \mathbf{Z} \quad (2.2) \]

where each column vector of the $D_c \times K_c$ matrix $\mathbf{A} = [\mathbf{a}_1, ..., \mathbf{a}_{K_c}]$ is the pilot sequence of respective user, where $\Gamma$ is the $K_c \times K_c$ diagonal matrix satisfying $\Gamma_{kk} = \gamma_k, \forall k \in [K_c]$ where in turn $\gamma_k = b_k g_k, \forall k \in [K_c]$ is the actual transmitted signal power for a user $k$, where $\mathbf{H} = [\mathbf{h}_1, ..., \mathbf{h}_{K_c}]^T$, and where $\mathbf{Z}$ is the $D_c \times M$ matrix satisfying $\mathbf{Z}_{:i} = \mathbf{z}[i]$. We define the average signal to noise ratio (SNR) of a user $k$ as

\[ \text{SNR}_k = \frac{\gamma_k}{\sigma^2}, k \in \mathcal{A}_c \quad (2.3) \]

but leave the motivation for such a definition to [5].

### 2.4 The problem

The goal of activity detection is to, from the received signal $Y$ and the known pilot sequences $\mathbf{A}$, find an estimate $\hat{\gamma}$ of $\gamma$ (or $\hat{\Gamma}$ of $\Gamma$) such that the set of active users $\mathcal{A}_c := \{k \in [K_c] : \gamma_k > \nu \sigma^2\}$ can be estimated by $\hat{\mathcal{A}}_c := \{k \in [K_c] : \hat{\gamma}_k > \nu \sigma^2\}$ for some pre-specified threshold $\nu > 0$. An illustration of this shown in Figure 2.3.

Just like in [5] we define the metric probability of miss detection averaged over active users as

\[ p_{MD}(\nu) = 1 - \frac{\mathbb{E}[|\hat{\mathcal{A}}_c(\nu) \cap \mathcal{A}_c|]}{\mathcal{A}_c} \quad (2.4) \]

and the metric probability of false alarm averaged over inactive users as

\[ p_{FA}(\nu) = \frac{\mathbb{E}[|\hat{\mathcal{A}}_c(\nu) \setminus \mathcal{A}_c|]}{K_c - \mathcal{A}_c} \quad (2.5) \]

Further, we define a new metric, probability of error, as

\[ p_E = \min_{\nu > 0} \{\max\{p_{MD}(\nu), p_{FA}(\nu)\}\} \quad (2.6) \]

which will be used to study the convergence with respect to algorithm iterations.
2.5 Algorithm to implement

Coordinate descent is an algorithm used for finding the minimum (or maximum) point in smooth convex functions. By making an initial guess at a set of coordinates, the algorithm optimizes and moves the current estimate along one coordinate at a time, for each iteration, until convergence is reached (a minimum is found) or for a fixed number of iterations. In this work, coordinate descent is used to minimize a likelihood statistic estimate of $\gamma$ as described in [5].

The primary algorithm of coordinate descent for activity detection with maximum likelihood estimator was first described in [5] and can be seen in Algorithm 1. A slightly modified version with Sherman-Morrison matrix inversion [12] is described in [2] which utilizes matrix inverse updating during operation to surpass the need of explicitly evaluating matrix inverses for each iteration (Line 7 in Algorithm 2). Note that the Sherman-Morrison matrix inversion is not an estimate of a matrix inversion, but a derive mathematical theorem, making the output estimate $\hat{\gamma}$ of Algorithm 1 and Algorithm 2 exactly equivalent.

Notice that, on Line 3 in Algorithms 1 and 2, coordinate are selected at random. It is proven that selecting coordinates randomly give the algorithm a much greater convergence rate than updating coordinates in a static fashion, and it is therefore a common approach when constructing coordinate descent algorithms [4].

In Algorithms 1 and 2, input $\sigma^2$ should ideally be chosen (and is in this work chosen) according to

$$\sigma^2 = \frac{\gamma}{\text{SNR}}$$

(2.7)

where $\gamma = \gamma_k, \forall k \in A_c$ and $\text{SNR} = \text{SNR}_k, \forall k \in A_c$. Further, $I_c \in \mathbb{N}$ denotes the desired number of iterations to perform in the algorithm.

2.5.1 Updating the variance estimate

As can be seen, the only difference between Algorithms 1 and 2 is the usage of the Sherman-Morrison matrix inversion[12] (Algorithm 2, Line 7) to update the inverse of a variance estimate component $\Sigma$. This approach of updating the inverse makes sense from an implementation point of view since only the inverse of $\Sigma$ is used in the algorithm. It is used as an intermediate value only to calculate the real algorithm output. Even more importantly, from an implementation perspective the explicit inverse of a full matrix is both computationally expensive and comes with a lot of latency, possibly slowing down an implementation.

2.5.2 Performance requirements

Requirements on the implementation performance are used to steer development in a direction that will make the implementation as useful as possible. The requirements used for this project is a fusion of requirements used in previous works ([5], [2]) as well as feedback given from Prof. Erik G. Larsson from the Division of Communication Systems in the Department of Electrical Engineering at Linköping University. The requirements on the system are displayed in Table 2.1.
Note especially that the parameter $A_c$ is the number of active users at any given time, i.e., all users can be active as long as no more than $A_c$ users are active simultaneously. With the Internet of Things being one of the main focus in 5G-MIMO technology, it makes sense to assume that only a fraction of all possible users are active at any given moment. The algorithm operating frequency $f$ is chosen such that the demand for low latency, which comes with 5G, can be satisfied.

2.6 Hardware aspects of implementation

This section will describe some theory behind the implementation. When taking an algorithm from its mathematical representation to one of its hardware implementation there are many things to consider such as numeric representations, word lengths, timings and much more.

2.6.1 Numeric representation

In every hardware implementation, each arithmetic operation and intermediate data storage is performed with some numeric representation suitable for the algorithm in question. We select such representations based on the dynamic range and numerical precision required for application performance and functionality.

In this work, the designer has chosen the fixed-point number representation to represent real data in the implementation. This numeric representation can be beneficial due to its simplicity with easy to vary dynamic range, easy to vary precision and its possibility to reuse many arithmetic operators, even when the radix of a representation changes. An example showing how fixed point is used to represent a fractional number is shown in Figure 2.4.

![Figure 2.4: Example of the number 3.25 represented in fixed-point with 4 integer bits and 4 fractional bits. The notation is often noted Q(4,4).](image)

2.6.2 Word length

Using any numeric representation with finite word length introduces error through quantization to the implementation [9]. With sufficient dynamic range and precision, this error can be kept at a minimum which is desired, but a longer word length also give rise to bigger implementations (more silicon area on ASIC) and a higher power consumption, both which are crucial to keep at a minimum. One task of implementing an ASIC is to find a set of word lengths wide enough to keep algorithmic performance within the requirements while also keeping the word lengths down to reduce chip area and power consumption.
In this work, a word length sweep of each internal node will be conducted to evaluate algorithmic performance vs word length. It is desirable to keep the performance roughly within the double-precision floating-point simulated performance.

2.6.3 Combinational latency

An application specific integrated circuit is, as the name suggests, an electrical device with some specific purpose implemented on an integrated circuit, usually in some CMOS process (in this case a 28-nm FD-SOI CMOS process). Parasitic electrical phenomenons, such as parasitic capacitance and inductance, will limit the propagation speed of electrical signals within the circuit. The time it takes for an electrical signal to propagate from one node in the system to another node (time taken between stimulation and response) is called latency. The time it takes for an electrical signal to propagate from one flip-flop to another, through some combinational logic, is called combinational latency.

2.6.4 Critical path

When making high performance circuits, the concept of latency is uttermost crucial to take into consideration when designing, since the greatest combinational latency in the system, the critical path, will limit the maximum clock frequency under which the implementation will function properly, and therefore put a limit on how fast the implementation can operate. The maximum combinational latency of the system $T_{\text{max}}$ will limit the maximum operating clock frequency $f_{\text{max}}$ according to

$$f_{\text{max}} = \frac{1}{T_{\text{max}}} \quad (2.8)$$

and an operating frequency greater than this theoretical maximum frequency will break intended synchronous behavior by using the result in a node before it is ready.

Loop free critical paths are however rather easy to accelerate. By introducing pipelining stages (usually D flip-flops) in the critical path, the combinational latency is effectively divided into two new, shorter paths. The maximum combinational latency in the system, the critical path, is then reduced allowing for a higher operating frequency. An example of this technique is illustrated in Figure 2.5.

2.6.5 Critical loop

When all none-looped paths have had their latency reduced such that the critical path is located inside of a feedback loop, it is no longer possible to shorten the critical path (and thereby increase operating frequency) without altering the behaviour of the synchronous digital system. When the critical path is located in a feedback loop we refer to the loop as the critical loop.
Figure 2.5: Example of a non-looped path in which the critical path could be reduced by introducing a D flip-flop between combinatorial networks $C_1$ and $C_2$.

Figure 2.6: Example of a system with feedback loop. Unlike in Figure 2.5, introducing a D flip-flop between combinatorial networks $C_1$ and $C_2$ will alter the synchronous behavior of the system.

This observation, as illustrated in Figure 2.5, will put a hard upper limit on the possible system operating frequency, by the means of limiting $T_{\text{max}}$ according to

$$\frac{1}{f_{\text{max}}} = T_{\text{max}} = \sum_{i \in \mathcal{C}} T_i / |\mathcal{A}|$$

(2.9)

where $\mathcal{C}$ is the set of combinational blocks in the critical loop, $T_i$ is the combinational latency of block $i \in \mathcal{C}$ in the loop and $\mathcal{A}$ is the set of pipeline (D flip-flops) elements in the critical loop. When the maximum system latency is located in the critical loop, we refer to $T_{\text{max}}$ as iteration period bound.

To alter this hard upper frequency limit one is required to perform some sort algorithmic transformation, that is, transform the algorithm in a way that will still produce the same result, but can be mathematically expressed in new way. Equation (2.10) shows a mathematical expression that can be used to transform an algorithm without altering the algorithmic result. Using such a transformation to the algorithm in its critical loop can potentially lower the iteration period bound since it requires less arithmetic operations, i.e., less combinational blocks. These algorithmic transformations are commonly more difficult to achieve, and is often the result of reworking an algorithm.

$$\max([|a + b|, |a - b|]) = |a| + |b|, a \in \mathbb{R}, b \in \mathbb{R}$$

(2.10)
2.6.6 Isomorphic implementation

An isomorphic implementation (also known as injective implementation or 1-to-1 mapping) is a way of implementing an algorithm in which each operation in the algorithm is mapped to an individual process element such that there exists a 1-to-1 mapping between each algorithm operation and each process element. Figure 2.7 tries to illustrate this concept. What makes an isomorphic implementation interesting is the observation that, for any given algorithm (that would not undergo any algorithm transformation), the isomorphic implementation will have the least amount of operational overhead and thereby be the fastest implementation. It is usually also the implementation with lowest power consumption per performance, but the isomorphic implementations usually consumes huge area compared to an implementation that reuses process elements for different algorithm operations, and will therefore be very expansive to fabricate.

Due to the fact that no previous implementation of this algorithm were found during the research phase of this project, and due to the difficulties in estimating the iteration period bound of an implementation of Algorithm 2, it makes sense to make a first attempt at an isomorphic implementation of the algorithm to see if it will be possible to make an implementation that will meet the throughput/latency performance requirements without undergoing any algorithm transformations. It was clear very early that the iteration period bound of the algorithm will be large (see flow graph, Figure 3.8) and it is therefore uncertain if it is possible to satisfy the requirements on latency and throughput of the system.

![Figure 2.7: Example illustrating two implementations of the same algorithm as a) isomorphic implementation where there exists one process element for evaluating each algorithm step and b) polymorphic implementation where a programmable process element is reused for the three different stages and controlled through some control unit.](image)
Algorithm 1: Activity detection via Coordinate Descend

**Input:** The \([D_c \times M]\) matrix \(Y\), the \([D_c \times K_c]\) matrix \(A\) and the noise variance \(\sigma^2\).

**Output:** The resulting signal power estimate \(\hat{\gamma} = [\hat{\gamma}_1, ..., \hat{\gamma}_{K_c}]^T\).

1. Let: \(\Sigma = \sigma^2 I_{D_c}\), \(\hat{\gamma} = 0\), \(\hat{\Sigma}_y = \frac{1}{M} YY^H\) and \(I_c \in \mathbb{N}\)
2. for \(i = 1, 2, ..., I_c\) do
   3. Let: The \([K_c \times 1]\) vector \(s\) be a random permutation of \([1, 2, ..., K_c]\).
   4. for \(k = s_1, s_2, ..., s_{K_c}\) do
      5. Let: \(d^* = \max \left\{ \frac{a_k^H \hat{\Sigma}_y \Sigma^{-1} a_k - a_k^H \Sigma^{-1} a_k}{(a_k^H \Sigma^{-1} a_k)^2}, -\gamma_k \right\}\)
      6. Set: \(\hat{\gamma}_k \leftarrow \hat{\gamma}_k + d^*\)
      7. Set: \(\Sigma \leftarrow \Sigma + d^*(a_k a_k^H)\)
   8. end
9. end

Algorithm 2: Activity detection via Coordinate Descend with Sherman-Morrison matrix inversion

**Input:** The \([D_c \times M]\) matrix \(Y\), the \([D_c \times K_c]\) matrix \(A\) and the noise variance \(\sigma^2\).

**Output:** The resulting signal power estimate \(\hat{\gamma} = [\hat{\gamma}_1, ..., \hat{\gamma}_{K_c}]^T\).

1. Let: \(\Sigma^{-1} = \frac{1}{\sigma^2} I_{D_c}\), \(\hat{\gamma} = 0\), \(\hat{\Sigma}_y = \frac{1}{M} YY^H\) and \(I_c \in \mathbb{N}\)
2. for \(i = 1, 2, ..., I_c\) do
   3. Let: The \([K_c \times 1]\) vector \(s\) be a random permutation of \([1, 2, ..., K_c]\).
   4. for \(k = s_1, s_2, ..., s_{K_c}\) do
      5. Let: \(d^* = \max \left\{ \frac{a_k^H \hat{\Sigma}_y \Sigma^{-1} a_k - a_k^H \Sigma^{-1} a_k}{(a_k^H \Sigma^{-1} a_k)^2}, -\gamma_k \right\}\)
      6. Set: \(\hat{\gamma}_k \leftarrow \hat{\gamma}_k + d^*\)
      7. Set: \(\Sigma^{-1} \leftarrow \Sigma^{-1} + d^* \frac{\Sigma^{-1} a_k a_k^H \Sigma^{-1}}{1 + d^* a_k^H \Sigma^{-1} a_k}\)
   8. end
9. end

**Table 2.1:** Requirements put on the system and used in simulation.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Interpretation</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>(K_c)</td>
<td>Maximum possible connected users</td>
<td>2048</td>
</tr>
<tr>
<td>(A_c)</td>
<td>Maximum simultaneous active users</td>
<td>200</td>
</tr>
<tr>
<td>(M)</td>
<td>Base station antennas</td>
<td>196</td>
</tr>
<tr>
<td>(L)</td>
<td>Pilot sequence length</td>
<td>100</td>
</tr>
<tr>
<td>(f)</td>
<td>Algorithm operating frequency</td>
<td>1000 Hz</td>
</tr>
<tr>
<td>SNR</td>
<td>Signal to noise ratio</td>
<td>10 dB</td>
</tr>
</tbody>
</table>
This chapter will present the results and findings of the isomorphic implementation of Algorithm 2. It will start of by describing the four different models that were created and used to verify the behaviour of the system. After that, the chapter will present some techniques used to reduce the resulting silicon area and possibly increase the operating frequency.

3.1 Models

To generate a final working and somewhat modular implementation, which should be easy to verify, models of increasing detail were generated until a final RTL model could be produced. For each new model of increasing detail, the previous model was used as a reference model for verifying the behavior of the new system.

Four different models were produced and an overview of the models is shown in Figure 3.1. Some of the different levels of detail for the four models is presented in Table 3.1. The final RTL model of the system is made with VHDL, from which a synthesis tool will generate a 28 nm FD-SOI ASIC description such that area and power consumption can be analyzed.

Table 3.1: The four models and their different levels of detail. C++$_1$ is the C++ floating-point model and C++$_2$ is the C++ fixed-point model.

<table>
<thead>
<tr>
<th></th>
<th>MATLAB</th>
<th>C++$_1$</th>
<th>C++$_2$</th>
<th>VHDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word length effects</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Operation control</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>RTL control</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
3.1.1 MATLAB model

An initial MATLAB high level simulation model had previously been made by the Division of Communication Systems to assess the algorithms statistical properties compared to some other activity detection algorithms. This MATLAB model served as the starting point for this implementation and was used extensively for testing modifications of the model to ease the work later in the more detailed models. The code for this model is attached to Appendix B.

It was also in the MATLAB model that the first word length simulations were added. Even though the underlying numeric data type used in MATLAB to represent real numbers is the IEEE 754 binary64 type (commonly known an double-precision floating-point), it was possible to introduce some finite word length effects through the use of emulated fixed point. Emulating the use of fixed-point number helped get a first glance of what word lengths to use in the later more detailed models. The fixed point emulation is shown in Figure 3.2.

However, since all matrix operations performed in the MATLAB model are still evaluated using double-precision floating-point arithmetic, the result will differ from that of an implementation where all operations, including the internal matrix/vector sub-operations, are performed entirely with fixed point numbers. Therefore, this model can not be used for verifying the later RTL-VHDL model of the implementation, but it is useful in that it is easy to modify and its results can be used as reference for other models.

![Figure 3.2: Fixed point emulation block used in the MATLAB model. The block will evaluate the double-precision floating-point number closest to input x represented as a fixed-point number with n fractional bits.](image-url)
3.1 Models

Figure 3.3: Illustration of a) general complex multiplier and b), c) conjugated multiplication using the same multiplier block.

3.1.2 C++ model using floating-point arithmetic

After the initial MATLAB model had been created, a first C++ model was created to obtain control over each and every operation in the algorithm implementation. In this model, the concept of matrices are removed and each matrix operation have to be manually performed on all sets of data. This model is, however, still using floating-point arithmetic.

In this model it is possible to eliminate the explicit hermitian operations from the algorithm by rearranging inputs to succeeding general complex multiplier such that a conjugated multiplication are performed. How inputs should be rearranged to a general complex multiplier to accomplish this is illustrated in Figure 3.3.

Furthermore, in this model it was possible to convert all multiplications with vector \( a_{\cdot k} \) to conditional propagation blocks. This is because matrix \( A \) satisfies \( A_{i,j} \in \{1, -1, -i, i\} \) which makes it possible to substitute multiplications with elements in the matrix with the block illustrated in Figure 3.4, which will consume less area and have a lower latency.

Figure 3.4: Block used to multiply a complex number with \( a \), where \( a \) satisfies \( a \in \{1, -1, -i, i\} \). Conditional carry-in on following adder is needed to realize the negation.
3.1.3 C++ model using fixed-point arithmetic

The second C++ model was based on the first C++ model but the underlying numeric data type was changed from floating-point numbers to fixed-point. This C++ model can be tested against previous reference models to verify algorithmic performance and then be used to verify the final RTL-VHDL model.

It was in this model that the choices for word lengths of each operation was finalized. This model served as the link between the difficult to modify RTL model and the very modular initial models.

3.1.4 RTL VHDL model

After the three earlier models had been created, it was rather straight forward to create the VHDL model. The goal when making this model was to make it do exactly what the previous model, the C++ model using fixed point arithmetic, was doing such that the VHDL model could be verified against it. This model will also be synthesizable making it possible to generate an ASIC.

Due to the complexity of making high level optimizations to the algorithm with this model, no new optimizations were introduced in this model. The verification of this model is described under Section 3.4.

3.2 Sequence

The algorithm as it is described in [2] and [5] propose updating users estimated channel strength at random, one time per coordinate descent iteration (Algorithm 2, Line 3). With the given communication model described under Section 2.3 and the requirements outlined in Table 2.1, the resulting algorithmic performance of the algorithm is shown in Figure 3.5.

With this method of selecting users at random within each coordinate descent iteration, one can see that the maximum algorithmic performance is reached within five, or no more than six coordinate descent iterations (left plot, Figure 3.5). For the sake of implementation we would like to avoid implementing an advanced random number permutator into the implementation. It is desirable to find a way of selecting users according to some scheme such that the algorithm convergence is still like, or close to, that of the fully randomized user sequence presented in Figure 3.5. A user selection scheme whose convergence is not fast enough will force the implementation to evaluate more coordinate descent iterations to attain the same algorithmic performance which will result in a slower and bigger implementation.

3.2.1 Random sequences with memories

At first, the idea of pre-generating some random sequence and than reusing those sequences between iterations or between algorithm runs were tested. This, however, is undesirable in that it requires memories to store the random sequences for usage. The result is presented in Figure 3.6.
Figure 3.5: Performance of the algorithm when evaluated exactly according to Algorithm 2, that is, with users updated at random exactly one time every coordinate descent iteration. Result is produced with the MATLAB model. Simulation settings are shown in Table 2.1 and the curves are produced with 250 000 Monte Carlo simulations.

Figure 3.6: Performance of the algorithm a) when seven random sequences are reused for respective algorithm iteration and b) were one random sequence is reused between all iterations.
The strategy of pre-generating six or seven random sequences and reusing these sequences shows promising results [a) in Figure 3.6], however, since this method of selecting users will require memories to store the sequences, it is better to try find a sequence that can be generated at runtime, that will not have to utilize memories (or advanced random number permutators).

It is worth mentioning that reusing one random sequence between all algorithm iterations [b) in Figure 3.6] converges as slow as updating users according to the sequence \{1, 2, 3, ..., \(K_c\)\} in all iterations. This seems to indicates that, for good convergence, it is more important to have sequences that vary greatly between iteration than it is to have randomness within each iteration.

### 3.2.2 Pseudo random modulo sequence

A pseudo random permutation that can be generated on the fly, without memories, that makes the algorithm reach its minimum probability of error within five or six iteration, is desirable. Equation

\[
k_{i,j} = (2i + 1)j \mod K_c, \; i \in \{0, ..., I_c - 1\}, \; j \in \{0, ..., K_c - 1\}
\]

(3.1)
can be used to generate permutations of the set \{0, 1, 2, ..., \(K_c - 1\)\}. In (3.1), \(I_c\) is used to denote the desirable number of coordinate descent iterations to perform and \(K_c\) the maximum number of possible users in the system.

Selecting users according to this equation will, from an implementation perspective, be a good choice. It requires two counters, one for counting coordinate descent iteration and one for counting users within an interation, plus some simple arithmetic, an adder plus two multipliers. If choosing \(K_c\) as a power of two (as in the requirements, Table 2.1) the modulo operation can be performed by using the \(\log_2(K_c)\) least significant bits in the result of \((2i + 1)j\).

The matrix

\[
\begin{pmatrix}
k_{0,0} & k_{0,1} & k_{0,2} & \cdots & k_{0,K_c-1} \\
k_{1,0} & k_{1,1} & k_{1,2} & \cdots & k_{1,K_c-1} \\
k_{2,0} & k_{2,1} & k_{2,2} & \cdots & k_{2,K_c-1} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
k_{I_c,0} & k_{I_c,1} & k_{I_c,2} & \cdots & k_{I_c-1,K_c-1}
\end{pmatrix}
\begin{pmatrix}
0 & 1 & 2 & \cdots & K_c - 1 \\
0 & 3 & 6 & \cdots & K_c - 3 \\
0 & 5 & 10 & \cdots & K_c - 5 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
0I_c & 1I_c & 2I_c & \cdots & K_c - 2I_c - 1
\end{pmatrix}
\]

(3.2)

visualizes how users are selected according to this scheme. Note that, in each new coordinate descent iteration, every user is selected exactly once. Each row in (3.2) represents one iteration of coordinate descent and each column represent one user.

The resulting performance of using this strategy is shown in Figure 3.7. In it we can see that the convergence of the algorithm is a somewhat slower for the first few iterations than compared to the algorithm using fully random user selection. However it can be seen that the modulo random strategy still reaches the maximum performance (minimum probability of error) for five to six iteration. Since the goal is to reach best possible performance, it does not matter that it converges a little slower for the first few iterations, as long as the maximum performance is reached around the same iteration.
3.3 Implementation flow graphs

Flow graphs are used to visualize the flow of information for some system. It is a great first step in making a hardware description of that particular system. This section will present the flow graphs used to visualize the information flow of Algorithm 2. Figure 3.8 shall perform one user iteration of line 4 in the algorithm. It is performing one user iteration of the coordinate descent algorithm. Figure 3.9 shows the top module of the coordinate descent implementation.

Figure 3.8 defines the node names of different nodes within the system. These node names are referenced later in the section about word lengths, Section 3.5. In the figure we can see that a lot of arithmetic operations are involving matrices which requires a lot of computational power, if one desires to evaluate them in parallel.

Figure 3.9, the top module of the system, is rather simple. It stores the covariance matrix $\Sigma^{-1}$ between coordinate descent user iterations and it is feed with signals coming from the outside.

3.4 Verification

The third generated model, the C++ model using fixed point arithmetic, was used to verify the fourth and final VHDL-RTL model. Since all random inputs of the algorithm have been eliminated, and both models are using fixed point arithmetic, it is possible to use the third model to verify exactly, model-to-model, the behavior of the final RTL-model. This method of verifying internal nodes model-to-model made the process of testing and readjusting the VHDL code rather quick.
Figure 3.8: Flow graph of the coordinate descent block. Asterisks (*) denote left hand side of operator when appropriate. \([a \times b]\) denote a matrix of 'a' rows and 'b' columns. Names on the arcs (e.g, \(S_2\)) are used to name nodes.

Figure 3.9: Flow graph of the coordinate descent top module. CD is a block of the flow graph in Figure 3.8.
3.5 Word length simulations

A set of word length simulations was conducted in an attempt to optimize word lengths for retaining the maximum algorithmic performance while minimizing the word lengths (more under Section 2.6.2). The data in each node is quantized to be represented by some number of fixed point binary and these were in turn swept.

#### 3.5.1 Integer bits

By extracting and visualizing node values when data flows through the system, it is possible to run simulations and than extract the maximum value for each node. This maximum value for some node, $a_{\text{max}} \in \mathbb{R}$, can be used to get a quantitative measurement of the required integer bits, $n_{\text{int}}$, for the implementation according
3 Results of isomorphic implementation

to \( n_{\text{int}} = \lceil \log_2(a_{\text{max}}) \rceil \) if \( a_{\text{max}} \geq 0 \) (i.e., the number can be represented using an unsigned fixed-point number) or \( n_{\text{int}} = \lceil \log_2(|a_{\text{max}}| + 1) \rceil + 1 \) otherwise (i.e., the number needs a signed fixed-point representation). This methodology was used to get an initial idea of how many integer bits were required to not overflow a node.

Further, in the MATLAB model and both the C++ models, a system was put in place which will warn the user if a node over- or underflows, making it possible to adjust (increase) the integer word length in the early models. Finally, with the word lengths presented in Appendix A, a 100 000 Monte Carlo simulation was performed to test that no node would over- or underflow.

3.5.2 Fractional bits

The required number of fractional bits was attained by running a sweeping simulating for different word lengths in the system. Each node was swept, one at a time, from an obvious plenitude of fractional bits, in a decreasing manner, until the algorithmic performance was visibly degraded. The minimum fractional word length that did not affect the algorithmic performance was chosen. This was done for all nodes in the system.

The results of these word length sweeps are shown in Appendix A. In it we can see that the required number of fractional bits varies from 0 up to 25, depending on which node is being studied.

3.6 Synthesis

Due to the extensive amount of calculations needed to perform the algorithm, an isomorph implementation turned out being much to big for actual ASIC implementation. Even with 500 GiB of primary memory on the synthesis machine, the lack of memory hindered complete synthesis with both Cadence Genus Synthesis Solution and Synopsys Design Compiler, meaning that the final implementation could never be completed.

Even if this implementation could never be finalized and synthesized into an ASIC, the results and theory of this project were not in vain. Implementation of a non-isomorphic, time-divided architecture were attempted [7] after this thesis had ran its course, and in the latter attempt, the ASIC implementation turned out successful. That project [7] was a continuation of this project, and in it most of the results from this thesis could be used to speed up the process immensely.

3.7 Conclusion

This chapter has presented an implementation of the coordinate descent algorithm shown in Algorithm 2. Four different implementation was created, three in software and one in VHDL, and the VHDL model was synthesized to an ASIC in a 28-nm FD-SOI CMOS process. Unfortunately the size of the isomorphic im-
plementation hindered this implementation from succeeding, but a time-divided architecture was attempted [7] in which generating the ASIC proved successful.
This chapter will present some further findings of the project which can be of interest if acceleration of algorithm operation frequency is desirable. It will present a parallelizable version of the coordinate descent algorithm and we will through simulation motivate an almost non-existing degradation of algorithmic performance using the parallel algorithm.

Further, this chapter will show that the more general Woodbury matrix identity, as opposed to the Sherman-Morrison matrix inversion, can be used to reduce the size of the otherwise \([L \times L]\) large matrix inversion, even in the parallel version of the algorithm.

### 4.1 Parallel execution

A first impression of Algorithms 1 and 2 is that both algorithms are rather sequential in that they perform one Coordinate Descent update stage (Algorithm 1, Line 5-6) for some possible user \(k\) and then update the variance estimate component, \(\Sigma\) or \(\Sigma^{-1}\), before proceeding doing the same operation again for a new user \(k\), only this time with a new estimate of the variance component. This sequential outline of the algorithm puts a tight limit on the speed we can achieve on an implementation of the algorithm. Therefore it makes sense to explore a more parallel version of the algorithm, one that does not lose us any algorithmic performance while doing so. In [10] it is argued that “... coordinate descent methods can be accelerated by parallelization when applied to the problem of minimizing the sum of a partially separable smooth convex function and a simple separable convex function.” When applying such a technique to Algorithm 1 we get a variant in which the Coordinate Descent update stages can be performed in parallel to significantly increase the achievable speed of the algorithm. Through simulations we will motivate the non degradation of the algorithm for usage against
the non parallelized counterpart. The parallelized algorithm is described in Algorithm 3.

**Algorithm 3: Activity detection via Parallel Coordinate Descend**

**Input:** The \([D_c \times M]\) matrix \(Y\), the \([D_c \times K_c]\) matrix \(A\) and the noise variance \(\sigma^2\).

**Output:** The resulting signal power estimate \(\hat{\gamma} = [\hat{\gamma}_1, ..., \hat{\gamma}_{K_c}]^T\).

1. Let: \(\Sigma = \sigma^2 I_{D_c}, \hat{\gamma} = \bar{0}, \hat{\Sigma}_y = \frac{1}{M} YY^H, I_c \in \mathbb{N}\) and \(P = \text{degree of parallelism}\).

2. for \(i = 1, 2, ..., I_c\) do

3. Let: The \([K_c \times 1]\) vector \(s\) be a random permutation of \([1, 2, ..., K_c]\).

4. for \(j = 1, 2, ..., \frac{K_c}{P}\) do

5. parallel for \(k = s_1 + jP, s_2 + jP, ..., s_{K_c} + jP\) do

6. Let: \(d_k^* = \max\left\{\frac{a_k^H \Sigma^{-1} \Sigma^{-1} a_k - a_k^H \Sigma^{-1} a_k}{(a_k^H \Sigma^{-1} a_k)^2}, -\gamma_k\right\}\)

7. Set: \(\hat{\gamma}_k \leftarrow \hat{\gamma}_k + d_k^*\)

8. end

9. Set: \(\Sigma \leftarrow \Sigma + \sum_k d_k^*(a_k a_k^H)\)

10. end

11. end

A MATLAB high level model of the parallel coordinate descent algorithm was created and simulated. Simulations were performed with settings as described in Table 2.1, and the results can be seen in Figure 4.1. A quick reminder that \(P\) denotes the level of parallelism for the algorithm. In Figure 4.1 we can clearly see that the performance degradation is non existing for degrees of parallelism up to 256 out of 2048. For 512 users out of 2048 evaluated in parallel it can be seen that the algorithm breaks down. If given infinite silicon area for the implementation, the degree of parallelism \(P\) could potentially result in roughly a \(P\) times as fast implementation, since most of the latency resulted from the algorithm comes from this inner loop (Algorithm 3, Line 5).

### 4.2 Reducing the size of matrix inverse in parallel coordinate descent

The primary difference between Algorithm 1 and 2 can be found in Line 7 were the authors of [2] suggest updating the covariance component inverse \((\Sigma^{-1})\) instead of the covariance component \((\Sigma)\) through the use of Sheerman-Morrisons formula [12]. Computationally, this transforms an explicit \([L \times L]\) matrix inverse to some basic matrix arithmetic. Sheerman-Morrisons formula is shown in Equation 4.1

\[
\left(\Sigma + a_k d_k^* a_k^H\right)^{-1} = \Sigma^{-1} - d_k^* \frac{\Sigma^{-1} a_k a_k^H \Sigma^{-1}}{1 + d_k^* a_k^H \Sigma^{-1} a_k} \quad (4.1)
\]
However, for the parallel version of coordinate descent it is necessary to update \( \Sigma^{-1} \) according to

\[
\Sigma^{-1} \leftarrow \left( \Sigma + \sum_{p=1}^{P} a_p d_p^* a_p^H \right)^{-1} \tag{4.2}
\]

where \( P \) is the number of parallel users in coordinate descent to update in parallel. To evaluate the right hand side of (4.2) (without explicitly performing the inverse), one need turn to the more general Woodbury matrix identity [6] (from which Sherman-Morrison formula can be derived). Woodburys matrix identity, when using this works notation, can be expressed as

\[
(\Sigma + \mathbf{A}_{:,[k:k+p]} \mathbf{D} \mathbf{A}_{:,[k:k+p]}^H)^{-1} = \Sigma^{-1} - \Sigma^{-1} \mathbf{A}_{:,[k:k+p]} \left( \mathbf{D}^{-1} + \mathbf{A}_{:,[k:k+p]}^H \Sigma^{-1} \mathbf{A}_{:,[k:k+p]} \right)^{-1} \mathbf{A}_{:,[k:k+p]}^H \Sigma^{-1} \tag{4.3}
\]

where \( \mathbf{A}_{:,[k:k+p]} = [a_k \ a_{k+1} \ \ldots \ \ a_{k+p}] \) is the \([L \times p]\) matrix of pilot sequences for users \( \{k, k+1, k+p\} \subseteq \mathcal{K}_c \) and where \( \mathbf{D} = \text{diag}(d_k^*, d_{k+1}^*, \ldots, d_{k+p}^*) \iff \mathbf{D}^{-1} = \text{diag}\left(\frac{1}{d_k}, \frac{1}{d_{k+1}}, \ldots, \frac{1}{d_{k+p}}\right) \). With the following example

\[
(\Sigma + a_1 d_1^* a_1^H + a_2 d_2^* a_2^H + a_3 d_3^* a_3^H)^{-1} = (\Sigma + \mathbf{A}_{:,[1:3]} \mathbf{D} \mathbf{A}_{:,[1:3]}^H)^{-1} \tag{4.4}
\]

we can see how the identity can be used to evaluate the update of covariance matrix \( \Sigma^{-1} \) from three users being evaluated in parallel.

At first glance the result in (4.3) seems to bring more work than simplification. However, note that quite a bit of partial results in the right hand side have already previously been evaluated and can simply be reused. More specifically, all elements in the matrix product

\[
\Sigma^{-1} \mathbf{A}_{:,[k:k+p]} = \begin{bmatrix} \Sigma^{-1} a_k & \Sigma^{-1} a_{k+1} & \ldots & \Sigma^{-1} a_{k+p} \end{bmatrix} \tag{4.5}
\]
have already been evaluated previously in the algorithm, moreover, all elements in the matrix product

$$A^H_{:,[k:k+p]} \Sigma^{-1} = \begin{bmatrix} a^H_k \Sigma^{-1} & a^H_{k+1} \Sigma^{-1} & \cdots & a^H_{k+p} \Sigma^{-1} \end{bmatrix}^T$$  \hspace{1cm} (4.6)

have also previously been evaluated in the algorithm and furthermore the diagonal elements of matrix product

$$A^H_{:,[k:k+p]} \Sigma^{-1} A_{:,[k:k+p]} = \begin{bmatrix} a^H_k \Sigma^{-1} a_k & a^H_{k+1} \Sigma^{-1} a_{k+1} & \cdots & a^H_{k+p} \Sigma^{-1} a_{k+p} \\ a^H_{k+1} \Sigma^{-1} a_k & a^H_{k+1} \Sigma^{-1} a_{k+1} & \cdots & a^H_{k+1} \Sigma^{-1} a_{k+p} \\ \vdots & \vdots & \ddots & \vdots \\ a^H_{k+p} \Sigma^{-1} a_k & a^H_{k+p} \Sigma^{-1} a_{k+1} & \cdots & a^H_{k+p} \Sigma^{-1} a_{k+p} \end{bmatrix}$$  \hspace{1cm} (4.7)

have previously been evaluated. The only missing partial result from the product $$A^H_{:,[k:k+p]} \Sigma^{-1} A_{:,[k:k+p]}$$ is the non-diagonal elements of which each element requires an $$L$$ long vector multiplication.

Due to the fact that new updates of the signal strength estimates converge to zero ($$d^* \rightarrow 0$$) when the algorithm progresses, the matrix $$D = \text{diag}[d_k^*, d_{k+1}^*, \ldots, d_{k+p}^*]$$ will tend to singularity making the inverse $$D^{-1}$$ non-existent, rendering the Woodbury matrix identity (4.3) useless. To counteract this, we can use a slightly adjusted version of the this identity by looking at (18) in [6] which proposes a variant of Woodburys matrix identity for when the matrix $$D$$ is allowed to be singular, i.e.,

$$\Sigma^{-1} - \Sigma^{-1} A_{:,[k:k+p]} D \left(I + A^H_{:,[k:k+p]} \Sigma^{-1} A_{:,[k:k+p]} D\right)^{-1} A^H_{:,[k:k+p]} \Sigma^{-1} = \left(\Sigma + A_{:,[k:k+p]} D A^H_{:,[k:k+p]}\right)^{-1} \hspace{1cm} (4.8)$$

Interestingly, from [6] we can read the following text attached to that equation, "His (Harvilles) results are of particular use in the maximum likelihood estimation of variance components". This describes exactly what we are trying to accomplish using the Woodbury matrix identity.

Using (4.8) to update the covariance component inverse ($$\Sigma^{-1}$$) can reduce the job of performing an $$[L \times L]$$ matrix inverse to evaluating an $$[P \times P]$$ inverse ($$P$$ being number of users evaluated in parallel) plus some additional arithmetic. Flow graphs for evaluating these extra partial results are presented in Section 4.3.

Figure 4.2 shows an architecture of parallel coordinate descent using Woodburys matrix identity and an architecture for parallel coordinate descent using explicit matrix inverse. Note especially that, as the degree of parallelism $$P$$ increases, so does the size of the matrix inverse in the Woodbury variant, but for the explicit matrix inversion variant, the size of the matrix inverse is constant and equal to the pilot sequence length $$L$$. One can for sure say that it is never worth pursuing the Woodbury variant if the degree of parallelism is greater than pilot sequence length (if $$P > L$$).
4.3 Flowgraphs for Woodbury architecture

Using Woodburys matrix identity to evaluate the algorithm will reduce the operations needed in the coordinate descent block due to the fact that the Sherman-Morrison rank-1 update will be removed. The reduced coordinate descent block is presented in Figure 4.3. The required partial results can then be evaluated according the configuration presented in Figure 4.4. To assemble the entire Woodbury architecture one also needs an \([L \times L]\) matrix inverter, and with that the architecture can be organized according to Figure 4.2 b).

4.4 Conclusion

Using the presented parallel coordinate descent algorithm, Algorithm 3, one can potentially improve throughput requirements by reducing the long sequential loop of coordinate descent, and we show with simulation that the algorithm can be parallelized such that at least 256 users (out of 2048) can be evaluated in parallel.

However, we can see in Section 4.2 that this parallelization also renders the Sherman-Morrison matrix formula useless, requiering the implementaiton to explicitly perform a matrix inverse again. Using the proposed modified Woodbury architecture can however reduce the size of the matrix inverse from an \([L \times L]\) inverse to a \([P \times P]\) inverse, where \(L\) is the pilot sequence length and \(P\) is the degree of parallelism.
Investigation of parallel algorithm

\[ \sum^{-1} a_k \] \[ a_k^H \Sigma^{-1} a_k \] \[ a_k^H \Sigma^{-1} a_k \] \[ d_{k + 1}^* \] \[ v[k] \]

**Figure 4.3:** Signal flow graph showing the reduced coordinate descent block (CD block) that is required to implement the parallel Woodbury algorithm.

**Figure 4.4:** Signal flow graph showing four coordinate descent blocks being evaluated in parallel, and generation of the partial results needed for the Woodbury inverse (4.8).
5

Conclusion and final remarks

This chapter will briefly summarize and conclude the results of this project.

5.1 Algorithms

Two algorithms, Algorithms 1 and 2 which are proposed in [5] respective [2], are analyzed and tuned from a hardware perspective. In Chapter 3 we demonstrate some techniques which can help reduce the size and possibly increase operation frequency of these algorithms.

Due to the how sequential coordinate descent algorithms are, in Chapter 4, we propose a parallelized version of the coordinate descent algorithm, Algorithm 3, and through simulation we motivate its viability when compared with Algorithms 1 and 2. This altered algorithm have the potential to significantly increase operation frequency of an implementation, but it also comes with some new difficulties.

5.2 Implementation

An ASIC implementation in a 28-nm FD-SOI process which evaluates Algorithm 2 is produced. The implementation supports $K_c = 2048$ possible connected users, to a base station with $M = 196$ antennas and with user pilot sequence lengths of $L = 100$. Although the implementation is much to big for actual synthesis, the author later created a new time divided architecture, based on this work, in which implementation was successful[7].

In Section 3.4 we present a way of verifying the behavior of the implementation, through model-to-model verification. By making sure that all node values match between associated models, we verify the behaviour of the VHDL-RTL
model, which is finally synthesized to an ASIC implementation. Because of the model-to-model verification steps, we argue that the final resulting algorithmic performance of the ASIC implementation can be seen in Figure 3.7.
Appendix
This appendix will present Iteration vs Error and ROC simulation results for the work. In the figures presented in this appendix (Figure A.1-A.10), the legend name \([n, m]\) is used to denote a word length of \(n\) integer bits and \(m\) fractional bits (e.g., \([10,5]\) represents a word length of 10 integer bits and 5 fractional bits). The node names can be found in Figure 3.8.

The settings used to produce these results are shown in Table 2.1. The results are generated from Monte Carlo simulations of at least 25 000 simulations.

![Graphs showing Iteration vs Error and ROC simulation results](Figure A.1: Result of wordlength simulation in node \(\delta\).)
Figure A.2: Result of wordlength simulation in node $S_2$.

Figure A.3: Result of wordlength simulation in node $\hat{\gamma}$.

Figure A.4: Result of wordlength simulation in node $N_{1,\text{tmp}}$. 
Figure A.5: Result of wordlength simulation in node $N_2$.

Figure A.6: Result of wordlength simulation in node $N_2^2$.

Figure A.7: Result of wordlength simulation in node $q$. 
Figure A.8: Result of wordlength simulation in node $\Sigma_y$.

Figure A.9: Result of wordlength simulation in node $Val$.

Figure A.10: Result of wordlength simulation in node $YY^H$. 
This appendix will present the initial MATLAB model from which this work is based on. It was primarily written by Unnikrishnan Kunnath Ganesan and modifications have been applied by Oscar Gustafsson, both working at the Department of Electrical Engineering at Linköping University.

### B.1 File: "GenerateBinarySequence.m"

```matlab
function S = GenerateBinarySignatureSequence (L, K)

    % L : Signature Length
    % K : Number of Sequences
    S = (2*round(rand(L, K))-1 + 1i*(2*round(rand(L, K))-1))/sqrt(2);

    S = S*exp(1i*pi/4);
end
```

### B.2 File: "ComputePfaPmd.m"

```matlab
function [Pmd, Pfa] = ComputePfaPmd(gamma, gamma_hat, sigma_sq, Kc, Ac, thresh)

    Pd = [mean(sum(gamma>0 & gamma_hat>thresh))/Ac];
    Pfa = [mean(sum(gamma_hat>thresh) & ~(gamma>0))/((Kc-Ac))];

    Pmd = 1-Pd;
end
```
B.3 File: "GenerateComplexGaussian"

```matlab
function ret_val = GenerateComplexGaussian(row, col, var)
    ret_val = sqrt(var) * (1 / sqrt(2)) * complex(randn(row, col), randn(row, col));
end
```

B.4 File: "CoordinateDescend_AD_ML_Fast.m"

```matlab
function [gamma_hat] = CoordinateDescend_AD_ML_Fast(Y, A, sigma_sqr, M, Kc, L)
    % Initialize the coordinate descent method.
    gamma_hat = zeros(1, Kc); % Initialize gamma_hat.
    Sigma_Y = (1/M) * (Y * Y'); % Initialize Sigma_Y.
    Sigma_inv = (1/sigma_sqr) * diag(ones(1, L));

    for i = 1:1:10
        % Random permutation of Kc.
        ridx = randperm(Kc, Kc);
        for k = ridx
            s2 = Sigma_inv * A(:, k);
            s1 = s2';
            N1 = real(s1 * Sigma_Y * s2);
            N2 = real(s1 * A(:, k));
            Val = (N1 - N2) / (N2^2);
            delta = max(Val, -1 * gamma_hat(k));
            t = Kc * (i - 1) + k;
            gamma_hat(k) = gamma_hat(k) + delta;
            N3 = s2 * s1;
            N4 = 1 + delta * N2;
            q = delta / N4;
            Sigma_inv = Sigma_inv - q * N3;
        end
    end
end
```
M = 196 ; % No of Antennas at Base Station.
Kc = 2048 ; % No of Potential Users.
Ac = 200 ; % Active Users.
L = 100 ; % Pilot Dimension, also know as Dc.
SNR = 10 ; % Signal to noise ratio [dB].
sigma_sqr = 1/SNR ; % Noise variance.

% Signature Sequence.
Acand = { GenerateBinarySignatureSequence ( L , Kc ) } ;
A = Acand { 1 } ;

% Monte carlo simulations to perform.
monte = 500 ;

% Threshold values.
threshold = 0.005:0.005:1.0 ;

% Create gamma vectors for comparison.
gamma = zeros ( monte , Kc ) ;
gamma_hat = zeros ( monte , Kc ) ;

% Main simulation loop.
parfor i = 1:monte

% Fading Channel Model.
H = GenerateComplexGaussian ( Kc , M , 1 ) ;

% Active User List. Kc from a total of Ac.
Ac_list = randperm ( Kc , Ac ) ;

% Transmitted Signal.
v = zeros ( 1 , Kc ) ;
v ( Ac_list ) = 1 ; % Signal power = 1.
gamma ( i , :) = v ; % Assign Tx power to devices.
Gamma_sqrt = diag ( sqrt ( gamma ( i , :) ) ) ;
Z = GenerateComplexGaussian ( L , M , sigma_sqr ) ;
Y = A * Gamma_sqrt * H + Z ;

% Run simulation.
gamma_hat ( i , :) = CoordinateDescend_AD_ML_Fast ( Y , A ,
   sigma_sqr , M , Kc , L ) ;
end
% The data in gamma and gamma_are are swapped to make
% monte carlo loop parallelizable. Swap them back.
gamma = gamma’;
gamma_hat = gamma_hat’;

% Generate ROC curve.

[Pmd_cd, Pfa_cd] = ComputePfaPmd(\texttt{gamma}, \texttt{gamma_hat}, \texttt{sigma_sqr}, \texttt{Kc}, \texttt{Ac}, \texttt{threshold});
\texttt{loglog(}Pmd_cd’, Pfa_cd’, \texttt{’LineWidth’, 2});
\texttt{xlabel(}’Probability of Miss Detection’\texttt{);}
\texttt{ylabel(}’Probability of False Alarm’\texttt{);}
\texttt{xlim([1e−6 1e−0]); ylim([1e−6 1e−0]);}
\texttt{title(}’ROC’\texttt{); grid on;}

% Save figures and data from simulation.
\texttt{savefig(}’fig_result.fig’\texttt{);}


